注: 1.阅读本章前你需要先了解我的XmlBeanFactory源码分析那两篇文章

ApplicationContext是spring推荐的IOC容器,他是BeanFactory的增强版

解析入口测试代码:

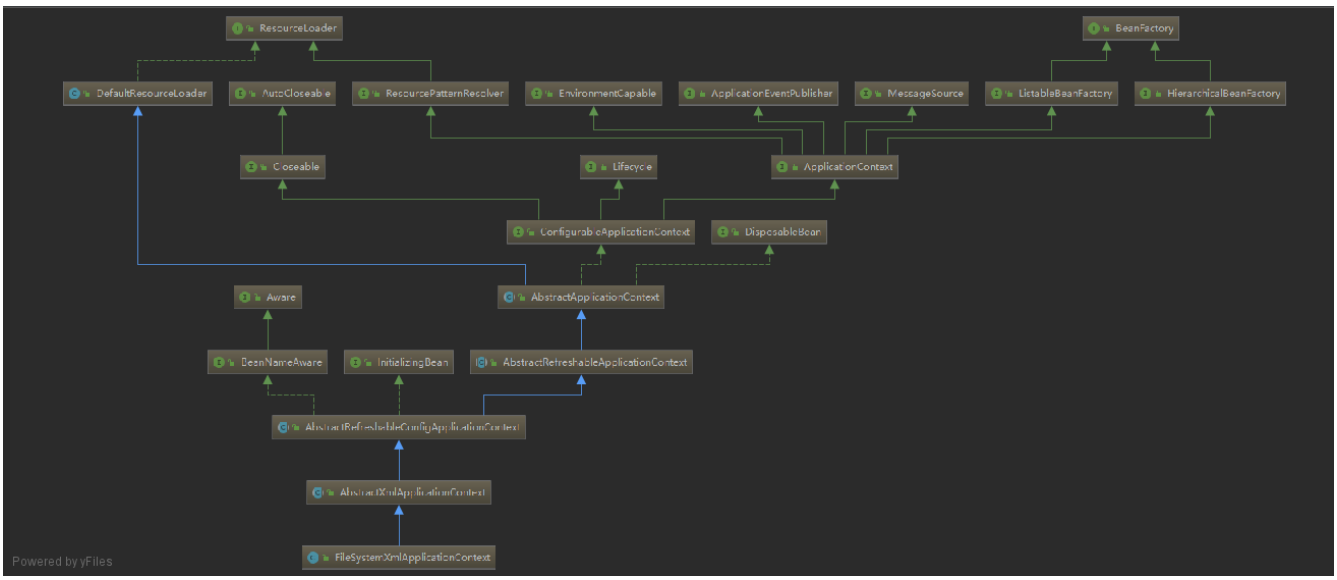
```
FileSystemXmlApplicationContext applicationContext = new
FileSystemXmlApplicationContext("classpath:/bean.xml");
Object person = applicationContext.getBean("person");
```

此处我在配置文件处建了一个bean.xml文件 里面主要配置了一个Perosn类的信息

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="person" class="com.mgw.bean.Person">
        <property name="age" value="10"></property>
        <property name="name" value="sansan"></property>
    </bean>
</beans>
```

先看几张类的继承关系图:



注:FileSystemXmlApplicationContext类的继承关系图

解析开始: 0.解析前的相关说明

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
ApplicationContext parent)
            throws BeansException {
    //为其父类AbstractApplicationContext做一些处理
    super(parent);
    //保存配置文件的路径信息
    setConfigLocations(configLocations);
    if (refresh) {
        //刷新  核心中的核心
        refresh();
    }
}
```

FileSystemXmlApplicationContext先给其父类AbstractApplicationContext做了一些处理

```
public AbstractApplicationContext(ApplicationContext parent) {
    this();
    setParent(parent);
}
public AbstractApplicationContext() {
    this.resourcePatternResolver = getResourcePatternResolver();
}
protected ResourcePatternResolver getResourcePatternResolver() {
    return new PathMatchingResourcePatternResolver(this);
}
public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {
    Assert.notNull(resourceLoader, "ResourceLoader must not be null");
    //将FileSystemXmlApplicationContext放入到PathMatchingResourcePatternResolver的resourceLoader中
FileSystemXmlApplicationContext继承了ResourceLoader接口
    this.resourceLoader = resourceLoader;
}
```

保存配置文件的路径信息:

```
public void setConfigLocations(String... locations) {
    if (locations != null) {
        Assert.noNullElements(locations, "Config locations must not be null");
        //配置文件的路径保存起来
        this.configLocations = new String[locations.length];
        for (int i = 0; i < locations.length; i++) {
            this.configLocations[i] = resolvePath(locations[i]).trim();
        }
    }
    else {
        this.configLocations = null;
    }
}
```

1.refresh()刷新 AbstractApplicationContext.refresh()这个类可谓是核心中核心 整个spring的全家桶中只要使用了核心的IOC容器这个类肯定需要无论是springMvc,springboot还是spring aop,tx(事务)还是springcloud都一定会使用这个类的refresh(),这个方法我们需要仔细说

```
public void refresh() throws BeansException, IllegalStateException {
```

```
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
            }

            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();

            // Reset 'active' flag.
            cancelRefresh(ex);

            // Propagate exception to caller.
            throw ex;
        }

        finally {
            // Reset common introspection caches in Spring's core, since we
```

```
            // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();
        }
    }
}
```

2.prepareRefresh()刷新前的预处理; 1）、initPropertySources()初始化一些属性设置;子类自定义个性化的属性设置方法; 2）、getEnvironment().validateRequiredProperties();检验属性的合法等 3）、earlyApplicationEvents= new LinkedHashSet();保存容器中的一些早期的事件;

```
protected void prepareRefresh() {
    this.startupDate = System.currentTimeMillis();
    this.closed.set(false);
    this.active.set(true);

    if (logger.isInfoEnabled()) {
        logger.info("Refreshing " + this);
    }

    // Initialize any placeholder property sources in the context environment
    //初始化一些属性设置;子类自定义个性化的属性设置方法
    initPropertySources();

    // Validate that all properties marked as required are resolvable
    // see ConfigurablePropertyResolver#setRequiredProperties
    //检验属性的合法等
    getEnvironment().validateRequiredProperties();

    // Allow for the collection of early ApplicationEvents,
    // to be published once the multicaster is available...
    //保存容器中的一些早期的事件
    this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
}
```

3、obtainFreshBeanFactory();获取BeanFactory

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    //使用委派模式调用它的子类(AbstractRefreshableApplicationContext)
    refreshBeanFactory();
    //返回一个bean工厂
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
    }
    return beanFactory;
}
//调用它的子类(AbstractRefreshableApplicationContext)
protected final void refreshBeanFactory() throws BeansException {
    //如果有bean工厂就销毁重新创建
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        //创建一个bean工厂(DefaultListableBeanFactory)
```

```
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        //使用委派模式调用它的子类(AbstractXmlApplicationContext)
        //载入bean的信息其实就是我的上篇XmlBeanFactory源码解析的上篇中说的bean的信息的定位,载入和解析以及注册这
三个过程
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition source for " +
getDisplayName(), ex);
    }
}
//载入Bean信息
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException,
IOException {
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    //创建一个XmlBeanDefinitionReader 是不是很熟悉?没错  就是我的上篇XmlBeanFactory源码解析中的过程了  只是略
有差别而已
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

    // Configure the bean definition reader with this context's
    // resource loading environment.
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    //将这个ApplicationContext传给ResourceEntityResolver实例 并将这个实例传给beanDefinitionReader  为后
面的doLoadDocument()中使用
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

    // Allow a subclass to provide custom initialization of the reader,
    // then proceed with actually loading the bean definitions.
    //初始化reader
    initBeanDefinitionReader(beanDefinitionReader);
    //载入bean
    loadBeanDefinitions(beanDefinitionReader);
}
//得到一个资源解析器
public ResourceEntityResolver(ResourceLoader resourceLoader) {
    super(resourceLoader.getClassLoader());
    this.resourceLoader = resourceLoader;
}
//使用reader去载入资源
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException,
IOException {
    //如果已经配置了Resource就直接拿出来并执行reader.loadBeanDefinitions(configResources); 实际上此
ApplicationC中是没有的
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);
    }
    //得到配置资源的路径
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        //载入资源与上篇XmlBeanFactory源码解析过程差不多，详情请见XmlBeanFactory源码解析(上)
```

```
        reader.loadBeanDefinitions(configLocations);
    }
}

//根据路劲去载入资源
public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws
BeanDefinitionStoreException {
    //拿到之前在步骤0中保存的ResourceLoader，就是那个this
    ResourceLoader resourceLoader = getResourceLoader();
    if (resourceLoader == null) {
        throw new BeanDefinitionStoreException(
                "Cannot import bean definitions from location [" + location + "]: no
ResourceLoader available");
    }

    if (resourceLoader instanceof ResourcePatternResolver) {
        // Resource pattern matching available.
        try {
            /*
            *public Resource[] getResources(String locationPattern) throws IOException {
            *   //使用之前步骤0创建好的PathMatchingResourcePatternResolver的getResources()方法
            *   return this.resourcePatternResolver.getResources(locationPattern);
            *}
            */
            Resource[] resources = ((ResourcePatternResolver)
resourceLoader).getResources(location);
            int loadCount = loadBeanDefinitions(resources);
            if (actualResources != null) {
                for (Resource resource : resources) {
                    actualResources.add(resource);
                }
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Loaded " + loadCount + " bean definitions from location pattern [" +
location + "]");
            }
            return loadCount;
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "Could not resolve bean definition resource pattern [" + location + "]", ex);
        }
    }
    else {
        // Can only load single resources by absolute URL.
        Resource resource = resourceLoader.getResource(location);
        int loadCount = loadBeanDefinitions(resource);
        if (actualResources != null) {
            actualResources.add(resource);
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Loaded " + loadCount + " bean definitions from location [" + location +
"]");
        }
        return loadCount;
    }
}
```

```
//拿到resources
public Resource[] getResources(String locationPattern) throws IOException {
    //使用之前步骤0创建好的PathMatchingResourcePatternResolver的getResources()方法
    return this.resourcePatternResolver.getResources(locationPattern);
}
//使用合适的方法创建Resource
public Resource[] getResources(String locationPattern) throws IOException {
    Assert.notNull(locationPattern, "Location pattern must not be null");
    if (locationPattern.startsWith(CLASSPATH_ALL_URL_PREFIX)) {
        // a class path resource (multiple resources for same name possible)
        if
(getPathMatcher().isPattern(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()))) {
            // a class path resource pattern
            return findPathMatchingResources(locationPattern);
        }
        else {
            // all class path resources with the given name
            return
findAllClassPathResources(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()));
        }
    }
    else {
        // Generally only look for a pattern after a prefix here,
        // and on Tomcat only after the "*/" separator for its "war:" protocol.
        int prefixEnd = (locationPattern.startsWith("war:") ? locationPattern.indexOf("*/") + 1 :
                locationPattern.indexOf(":") + 1);
        if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {
            // a file pattern
            return findPathMatchingResources(locationPattern);
        }
        else {
            // a single resource with the given name
            //因为我用的是"classpath:/bean.xml"所以走这里
            return new Resource[] {getResourceLoader().getResource(locationPattern)};
        }
    }
}
//"classpath:/bean.xml"配置的得到resource的方法
public Resource getResource(String location) {
    Assert.notNull(location, "Location must not be null");

    for (ProtocolResolver protocolResolver : this.protocolResolvers) {
        Resource resource = protocolResolver.resolve(location, this);
        if (resource != null) {
            return resource;
        }
    }

    if (location.startsWith("/")) {
        return getResourceByPath(location);
    }
    else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
        //返回一个ClassPathResource类型的Resource
        return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()),
getClassLoader());
    }
    else {
```

```
            try {
                // Try to parse the location as a URL...
                URL url = new URL(location);
                return new UrlResource(url);
            }
            catch (MalformedURLException ex) {
                // No URL -> resolve as resource path.
                return getResourceByPath(location);
            }
        }
    }
```
通过这些分析可以看到Resource的相关创建

至此bean信息的装配已经注册到了bean工厂中了

==========此时已经拿到了bean工厂==========

3、prepareBeanFactory(beanFactory);BeanFactory的预准备工作（BeanFactory进行一些设置

```
1）、设置BeanFactory的类加载器、支持表达式解析器...
2）、添加部分BeanPostProcessor【ApplicationContextAwareProcessor】
3）、设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xxx；
4）、注册可以解析的自动装配；我们能直接在任何组件中自动注入：
                BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext
5）、添加BeanPostProcessor【ApplicationListenerDetector】
6）、添加编译时的AspectJ；
7）、给BeanFactory中注册一些能用的组件；
        environment【ConfigurableEnvironment】、
        systemProperties【Map<String, Object>】、
        systemEnvironment【Map<String, Object>】
```

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        // Tell the internal bean factory to use the context's class loader etc.
        //1）、设置BeanFactory的类加载器、支持表达式解析器...
    beanFactory.setBeanClassLoader(getClassLoader());
        beanFactory.setBeanExpressionResolver(new
StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
        beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));

        // Configure the bean factory with context callbacks.
    //2）、添加部分BeanPostProcessor【ApplicationContextAwareProcessor】
        beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
    //3）、设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xxx；
        beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
        beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
        beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
        beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

        // BeanFactory interface not registered as resolvable type in a plain factory.
        // MessageSource registered (and found for autowiring) as a bean.
    //4）、注册可以解析的自动装配；我们能直接在任何组件中自动注入：
        beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
```

```
        beanFactory.registerResolvableDependency(ResourceLoader.class, this);
        beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
        beanFactory.registerResolvableDependency(ApplicationContext.class, this);

        // Register early post-processor for detecting inner beans as ApplicationListeners.
    //5)、添加BeanPostProcessor【ApplicationListenerDetector】
        beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

        // Detect a LoadTimeWeaver and prepare for weaving, if found.
    //6)、添加编译时的AspectJ;
        if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
                beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
                // Set a temporary ClassLoader for type matching.
                beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
        }

        // Register default environment beans.
    /*
    7)、给BeanFactory中注册一些能用的组件;
        environment【ConfigurableEnvironment】、
        systemProperties【Map<String, Object>】、
        systemEnvironment【Map<String, Object>】
        */
        if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
                beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
        }
        if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
                beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME,
getEnvironment().getSystemProperties());
        }
        if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
                beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().getSystemEnvironment());
        }
}
```

特别在此说一下ApplicationContextAwareProcessor这个后置处理器:

```
class ApplicationContextAwareProcessor implements BeanPostProcessor {

        private final ConfigurableApplicationContext applicationContext;

        private final StringValueResolver embeddedValueResolver;


        /**
         * Create a new ApplicationContextAwareProcessor for the given context.
         */
        public ApplicationContextAwareProcessor(ConfigurableApplicationContext applicationContext) {
                this.applicationContext = applicationContext;
                this.embeddedValueResolver = new
EmbeddedValueResolver(applicationContext.getBeanFactory());
        }

        /*
```

```
        当这个bean为EnvironmentAware,
        EmbeddedValueResolverAware,
        ResourceLoaderAware,
        ApplicationEventPublisherAware,
        MessageSourceAware,
        ApplicationContextAware时
                会进行相关的注入invokeAwareInterfaces()中执行
        */
        @Override
        public Object postProcessBeforeInitialization(final Object bean, String beanName) throws
BeansException {
                AccessControlContext acc = null;

                if (System.getSecurityManager() != null &&
                                (bean instanceof EnvironmentAware || bean instanceof
EmbeddedValueResolverAware ||
                                                bean instanceof ResourceLoaderAware || bean
instanceof ApplicationEventPublisherAware ||
                                                bean instanceof MessageSourceAware || bean
instanceof ApplicationContextAware)) {
                        acc = this.applicationContext.getBeanFactory().getAccessControlContext();
                }

                if (acc != null) {
                        AccessController.doPrivileged(new PrivilegedAction<Object>() {
                                @Override
                                public Object run() {
                                        invokeAwareInterfaces(bean);
                                        return null;
                                }
                        }, acc);
                }
                else {
                        invokeAwareInterfaces(bean);
                }

                return bean;
        }

        private void invokeAwareInterfaces(Object bean) {
                if (bean instanceof Aware) {
                        if (bean instanceof EnvironmentAware) {
                                ((EnvironmentAware)
bean).setEnvironment(this.applicationContext.getEnvironment());
                        }
                        if (bean instanceof EmbeddedValueResolverAware) {
                                ((EmbeddedValueResolverAware)
bean).setEmbeddedValueResolver(this.embeddedValueResolver);
                        }
                        if (bean instanceof ResourceLoaderAware) {
                                ((ResourceLoaderAware)
bean).setResourceLoader(this.applicationContext);
                        }
                        if (bean instanceof ApplicationEventPublisherAware) {
                                ((ApplicationEventPublisherAware)
bean).setApplicationEventPublisher(this.applicationContext);
                        }
```

```
                        if (bean instanceof MessageSourceAware) {
                                ((MessageSourceAware)
bean).setMessageSource(this.applicationContext);
                        }
                        if (bean instanceof ApplicationContextAware) {
                                ((ApplicationContextAware)
bean).setApplicationContext(this.applicationContext);
                        }
                }
        }

        @Override
        public Object postProcessAfterInitialization(Object bean, String beanName) {
                return bean;
        }

}
//以ApplicationContextAware为例进行说明
/*
Cat类实现了ApplicationContextAware这个接口，那么spring在创建bean这个类时会调用setApplicationContext这个方法自
动注入ApplicationContext
*/
public class Cat implements ApplicationContextAware {
        private ApplicationContext applicationContext;
        public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
                this.applicationContext = applicationContext;
        }
}
```

4、postProcessBeanFactory(beanFactory);BeanFactory准备工作完成后进行的后置处理工作； 1）、子类通过重写这个方法来在BeanFactory创建并预准备完成以后做进一步的设置

======================以上是BeanFactory的创建及预准备工作==================================

5、invokeBeanFactoryPostProcessors(beanFactory);执行BeanFactoryPostProcessor的方法； BeanFactoryPostProcessor这个接口与BeanPostProcessor其实很类似，BeanPostProcessor是拦截bean并增强bean的功能，那么BeanFactoryPostProcessor这个就是用来拦截BeanFactory的，并给其增强功能。 BeanFactoryPostProcessor：BeanFactory的后置处理器。在BeanFactory标准初始化之后执行的； 两个接口：BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor

注:BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子接口

执行BeanFactoryPostProcessor的方法； 先执行BeanDefinitionRegistryPostProcessor 1）、获取所有的BeanDefinitionRegistryPostProcessor； 2）、看先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor、 postProcessor.postProcessBeanDefinitionRegistry(registry) 3）、再执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor； postProcessor.postProcessBeanDefinitionRegistry(registry) 4）、最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors；postProcessor.postProcessBeanDefinitionRegistry(registry) 再执行BeanFactoryPostProcessor的方法 1）、获取所有的BeanFactoryPostProcessor 2）、看先执行实现了PriorityOrdered优先级接口的BeanFactoryPostProcessor、postProcessor.postProcessBeanFactory() 3）、在执行实现了Ordered顺序接口的BeanFactoryPostProcessor；postProcessor.postProcessBeanFactory() 4）、最后执行没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor； postProcessor.postProcessBeanFactory()

```
protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
    PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory,
getBeanFactoryPostProcessors());

    // Detect a LoadTimeWeaver and prepare for weaving, if found in the meantime
    // (e.g. through an @Bean method registered by ConfigurationClassPostProcessor)
    if (beanFactory.getTempClassLoader() == null &&
beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
        beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }
}
//执行beanFactory后置处理器方法
public static void invokeBeanFactoryPostProcessors(
                        ConfigurableListableBeanFactory beanFactory, List<BeanFactoryPostProcessor>
beanFactoryPostProcessors) {

        // Invoke BeanDefinitionRegistryPostProcessors first, if any.
        Set<String> processedBeans = new HashSet<String>();

        if (beanFactory instanceof BeanDefinitionRegistry) {
                BeanDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
                List<BeanFactoryPostProcessor> regularPostProcessors = new
LinkedList<BeanFactoryPostProcessor>();
                List<BeanDefinitionRegistryPostProcessor> registryProcessors = new
LinkedList<BeanDefinitionRegistryPostProcessor>();

                for (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {
                        if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
                                BeanDefinitionRegistryPostProcessor registryProcessor =
                                                (BeanDefinitionRegistryPostProcessor) postProcessor;
                                registryProcessor.postProcessBeanDefinitionRegistry(registry);
                                registryProcessors.add(registryProcessor);
                        }
                        else {
                                regularPostProcessors.add(postProcessor);
                        }
                }

                // Do not initialize FactoryBeans here: We need to leave all regular beans
                // uninitialized to let the bean factory post-processors apply to them!
                // Separate between BeanDefinitionRegistryPostProcessors that implement
                // PriorityOrdered, Ordered, and the rest.
                List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new
ArrayList<BeanDefinitionRegistryPostProcessor>();

                // First, invoke the BeanDefinitionRegistryPostProcessors that implement
PriorityOrdered.
        //先执行BeanDefinitionRegistryPostProcessor
        //获取所有的BeanDefinitionRegistryPostProcessor;
                String[] postProcessorNames =

beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
                for (String ppName : postProcessorNames) {
            /*
```

```
            看先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor、
                        postProcessor.postProcessBeanDefinitionRegistry(registry)
            */
                        if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                                currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                                processedBeans.add(ppName);
                        }
                }
                sortPostProcessors(currentRegistryProcessors, beanFactory);
                registryProcessors.addAll(currentRegistryProcessors);
                invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
                currentRegistryProcessors.clear();

                // Next, invoke the BeanDefinitionRegistryPostProcessors that implement Ordered.
                postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
                for (String ppName : postProcessorNames) {
            /*
            再执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor;
                        postProcessor.postProcessBeanDefinitionRegistry(registry)
            */
                        if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName,
Ordered.class)) {
                                currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                                processedBeans.add(ppName);
                        }
                }
                sortPostProcessors(currentRegistryProcessors, beanFactory);
                registryProcessors.addAll(currentRegistryProcessors);
                invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
                currentRegistryProcessors.clear();

                // Finally, invoke all other BeanDefinitionRegistryPostProcessors until no further
ones appear.
        /*
        最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors;
                postProcessor.postProcessBeanDefinitionRegistry(registry)
        */
                boolean reiterate = true;
                while (reiterate) {
                        reiterate = false;
                        postProcessorNames =
beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
                        for (String ppName : postProcessorNames) {
                                if (!processedBeans.contains(ppName)) {
                                        currentRegistryProcessors.add(beanFactory.getBean(ppName,
BeanDefinitionRegistryPostProcessor.class));
                                        processedBeans.add(ppName);
                                        reiterate = true;
                                }
                        }
                        sortPostProcessors(currentRegistryProcessors, beanFactory);
                        registryProcessors.addAll(currentRegistryProcessors);
                        invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
registry);
```

```
                        currentRegistryProcessors.clear();
                }

                // Now, invoke the postProcessBeanFactory callback of all processors handled so far.
                invokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
                invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);
        }

        else {
                // Invoke factory processors registered with the context instance.
                invokeBeanFactoryPostProcessors(beanFactoryPostProcessors, beanFactory);
        }

        // Do not initialize FactoryBeans here: We need to leave all regular beans
        // uninitialized to let the bean factory post-processors apply to them!
    //再执行BeanFactoryPostProcessor的方法
    //获取所有的BeanFactoryPostProcessor
        String[] postProcessorNames =
                        beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true,
false);

        // Separate between BeanFactoryPostProcessors that implement PriorityOrdered,
        // Ordered, and the rest.
        List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new
ArrayList<BeanFactoryPostProcessor>();
        List<String> orderedPostProcessorNames = new ArrayList<String>();
        List<String> nonOrderedPostProcessorNames = new ArrayList<String>();
        for (String ppName : postProcessorNames) {
                if (processedBeans.contains(ppName)) {
                        // skip - already processed in first phase above
                }
                else if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
            /*
            看先执行实现了PriorityOrdered优先级接口的BeanFactoryPostProcessor、
                        postProcessor.postProcessBeanFactory()
            */
                        priorityOrderedPostProcessors.add(beanFactory.getBean(ppName,
BeanFactoryPostProcessor.class));
                }
                else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                        /*
                        在执行实现了Ordered顺序接口的BeanFactoryPostProcessor;
                        postProcessor.postProcessBeanFactory()
                        */
            orderedPostProcessorNames.add(ppName);
                }
                else {
            /*
            最后执行没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor;
                        postProcessor.postProcessBeanFactory()
            */
                        nonOrderedPostProcessorNames.add(ppName);
                }
        }

        // First, invoke the BeanFactoryPostProcessors that implement PriorityOrdered.
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
```

```
        invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors, beanFactory);

        // Next, invoke the BeanFactoryPostProcessors that implement Ordered.
        List<BeanFactoryPostProcessor> orderedPostProcessors = new
ArrayList<BeanFactoryPostProcessor>();
        for (String postProcessorName : orderedPostProcessorNames) {
                orderedPostProcessors.add(beanFactory.getBean(postProcessorName,
BeanFactoryPostProcessor.class));
        }
        sortPostProcessors(orderedPostProcessors, beanFactory);
        invokeBeanFactoryPostProcessors(orderedPostProcessors, beanFactory);

        // Finally, invoke all other BeanFactoryPostProcessors.
        List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new
ArrayList<BeanFactoryPostProcessor>();
        for (String postProcessorName : nonOrderedPostProcessorNames) {
                nonOrderedPostProcessors.add(beanFactory.getBean(postProcessorName,
BeanFactoryPostProcessor.class));
        }
        invokeBeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

        // Clear cached merged bean definitions since the post-processors might have
        // modified the original metadata, e.g. replacing placeholders in values...
        beanFactory.clearMetadataCache();
}
```

举个例子:

```
@Component
public class MyBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProcessor{

        @Override
        public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws
BeansException {
                //postProcessBeanFactory()可以继续给bean工厂做功能增强
                System.out.println("MyBeanDefinitionRegistryPostProcessor...bean的数
量: "+beanFactory.getBeanDefinitionCount());
        }

        //BeanDefinitionRegistry Bean定义信息的保存中心，以后BeanFactory就是按照BeanDefinitionRegistry里面
保存的每一个bean定义信息创建bean实例;
        @Override
        public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws
BeansException {

                System.out.println("postProcessBeanDefinitionRegistry...bean的数
量: "+registry.getBeanDefinitionCount());
                //RootBeanDefinition beanDefinition = new RootBeanDefinition(Blue.class);
                //postProcessBeanDefinitionRegistry()可以给bean工厂继续注册bean信息
                AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(Blue.class).getBeanDefinition();
                registry.registerBeanDefinition("hello", beanDefinition);
        }

}
```

==========后面spring所有需要的bean都由bean工厂进行创建，具体过程参考我的XmlBeanFactory源码解析(下)=========

6、registerBeanPostProcessors(beanFactory);注册BeanPostProcessor（Bean的后置处理器）【 intercept bean creation】后置处理器已经创建(创建过程实际也是使用bean工厂创建对象详情见我的XmlBeanFactory源码解析(下)那篇博客)但是还未执行方法 不同接口类型的BeanPostProcessor；在Bean创建前后的执行时机是不一样的 BeanPostProcessor、DestructionAwareBeanPostProcessor、 InstantiationAwareBeanPostProcessor、SmartInstantiationAwareBeanPostProcessor、 MergedBeanDefinitionPostProcessor【internalPostProcessors】

```
1）、获取所有的 BeanPostProcessor;后置处理器都默认可以通过PriorityOrdered、Ordered接口来执行优先级
2）、先注册PriorityOrdered优先级接口的BeanPostProcessor；
把每一个BeanPostProcessor；添加到BeanFactory中
beanFactory.addBeanPostProcessor(postProcessor);
3）、再注册Ordered接口的
4）、最后注册没有实现任何优先级接口的
5）、最终注册MergedBeanDefinitionPostProcessor；
6）、注册一个ApplicationListenerDetector；来在Bean创建完成后检查是否是ApplicationListener，如果是
applicationContext.addApplicationListener((ApplicationListener<?>) bean);
```

```
public static void registerBeanPostProcessors(
                        ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext
applicationContext) {

    //获取所有的 BeanPostProcessor;后置处理器都默认可以通过PriorityOrdered、Ordered接口来执行优先级
        String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true,
false);

        // Register BeanPostProcessorChecker that logs an info message when
        // a bean is created during BeanPostProcessor instantiation, i.e. when
        // a bean is not eligible for getting processed by all BeanPostProcessors.
        int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 +
postProcessorNames.length;
        beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory,
beanProcessorTargetCount));

        // Separate between BeanPostProcessors that implement PriorityOrdered,
        // Ordered, and the rest.
        List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<BeanPostProcessor>();
        List<BeanPostProcessor> internalPostProcessors = new ArrayList<BeanPostProcessor>();
        List<String> orderedPostProcessorNames = new ArrayList<String>();
        List<String> nonOrderedPostProcessorNames = new ArrayList<String>();
        for (String ppName : postProcessorNames) {

                if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                        BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
                        priorityOrderedPostProcessors.add(pp);
                        if (pp instanceof MergedBeanDefinitionPostProcessor) {
                                internalPostProcessors.add(pp);
                        }
                }
                else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {

                        orderedPostProcessorNames.add(ppName);
                }
                else {
```

```
                        nonOrderedPostProcessorNames.add(ppName);
                }
        }

        // First, register the BeanPostProcessors that implement PriorityOrdered.
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
    /*
        先注册PriorityOrdered优先级接口的BeanPostProcessor;
                把每一个BeanPostProcessor; 添加到BeanFactory中
                beanFactory.addBeanPostProcessor(postProcessor);
     */
        registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

        // Next, register the BeanPostProcessors that implement Ordered.
        List<BeanPostProcessor> orderedPostProcessors = new ArrayList<BeanPostProcessor>();
        for (String ppName : orderedPostProcessorNames) {
                BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
                orderedPostProcessors.add(pp);
                if (pp instanceof MergedBeanDefinitionPostProcessor) {
                        internalPostProcessors.add(pp);
                }
        }
        sortPostProcessors(orderedPostProcessors, beanFactory);
    //再注册Ordered接口的
        registerBeanPostProcessors(beanFactory, orderedPostProcessors);

        // Now, register all regular BeanPostProcessors.
        List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<BeanPostProcessor>();
        for (String ppName : nonOrderedPostProcessorNames) {
                BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
                nonOrderedPostProcessors.add(pp);
                if (pp instanceof MergedBeanDefinitionPostProcessor) {
                        internalPostProcessors.add(pp);
                }
        }
    //最后注册没有标注优先级的PostProcessor;
        registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

        // Finally, re-register all internal BeanPostProcessors.
        sortPostProcessors(internalPostProcessors, beanFactory);
    //最终重新注册MergedBeanDefinitionPostProcessor; 保证其在最后面
        registerBeanPostProcessors(beanFactory, internalPostProcessors);

        // Re-register post-processor for detecting inner beans as ApplicationListeners,
        // moving it to the end of the processor chain (for picking up proxies etc).
    /*
    注册一个ApplicationListenerDetector; 来在Bean创建完成后检查是否是ApplicationListener, 如果是
        applicationContext.addApplicationListener((ApplicationListener<?>) bean);
    */
        beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));
}
```

7、initMessageSource();初始化MessageSource组件（做国际化功能；消息绑定，消息解析）； 1）、获取BeanFactory
2）、看容器中是否有id为messageSource的，类型是MessageSource的组件 如果有赋值给messageSource，如果没有自己创建一个DelegatingMessageSource； MessageSource：取出国际化配置文件中的某个key的值；能按照区域信息获取；
3）、把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource；
beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource);
MessageSource.getMessage(String code, Object[] args, String defaultMessage, Locale locale);

```
protected void initMessageSource() {
        //获取BeanFactory
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    /*
    看容器中是否有id为messageSource的，类型是MessageSource的组件
        如果有赋值给messageSource，如果没有自己创建一个DelegatingMessageSource；
        MessageSource：取出国际化配置文件中的某个key的值；能按照区域信息获取；
    */
        if (beanFactory.containsLocalBean(MESSAGE_SOURCE_BEAN_NAME)) {
                this.messageSource = beanFactory.getBean(MESSAGE_SOURCE_BEAN_NAME,
MessageSource.class);
                // Make MessageSource aware of parent MessageSource.
                if (this.parent != null && this.messageSource instanceof HierarchicalMessageSource)
{
                        HierarchicalMessageSource hms = (HierarchicalMessageSource)
this.messageSource;
                        if (hms.getParentMessageSource() == null) {
                                // Only set parent context as parent MessageSource if no parent
MessageSource
                                // registered already.
                                hms.setParentMessageSource(getInternalParentMessageSource());
                        }
                }
                if (logger.isDebugEnabled()) {
                        logger.debug("Using MessageSource [" + this.messageSource + "]");
                }
        }
        else {
                // Use empty MessageSource to be able to accept getMessage calls.
                DelegatingMessageSource dms = new DelegatingMessageSource();
                dms.setParentMessageSource(getInternalParentMessageSource());
                this.messageSource = dms;
        /*
        把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource；
                beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource);
                MessageSource.getMessage(String code, Object[] args, String defaultMessage, Locale
locale);
        */
                beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource);
                if (logger.isDebugEnabled()) {
                        logger.debug("Unable to locate MessageSource with name '" +
MESSAGE_SOURCE_BEAN_NAME +
                                        "': using default [" + this.messageSource + "]");
                }
        }
}
```

8、initApplicationEventMulticaster();初始化事件派发器； 1）、获取BeanFactory 2）、从BeanFactory中获取applicationEventMulticaster的ApplicationEventMulticaster； 3）、如果上一步没有配置；创建一个SimpleApplicationEventMulticaster 4）、将创建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件直接自动注入

```
protected void initApplicationEventMulticaster() {
    //获取BeanFactory
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    //从BeanFactory中获取applicationEventMulticaster的ApplicationEventMulticaster;
        if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
                this.applicationEventMulticaster =
                                beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
ApplicationEventMulticaster.class);
                if (logger.isDebugEnabled()) {
                        logger.debug("Using ApplicationEventMulticaster [" +
this.applicationEventMulticaster + "]");
                }
        }
        else {
        //如果上一步没有配置；创建一个SimpleApplicationEventMulticaster
                this.applicationEventMulticaster = new
SimpleApplicationEventMulticaster(beanFactory);
        //将创建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件直接自动注入
                beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
this.applicationEventMulticaster);
                if (logger.isDebugEnabled()) {
                        logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
                                        APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
                                        "': using default [" + this.applicationEventMulticaster +
"]");
                }
        }
}
```

9、onRefresh();留给子容器（子类） 1、子类重写这个方法，在容器刷新的时候可以自定义逻辑；

10、registerListeners();给容器中将所有项目里面的ApplicationListener注册进来； 1、从容器中拿到所有的ApplicationListener 2、将每个监听器添加到事件派发器中；
getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName); 3、派发之前步骤产生的事件；

```
protected void registerListeners() {
        // Register statically specified listeners first.
    //从容器中拿到所有的ApplicationListener
        for (ApplicationListener<?> listener : getApplicationListeners()) {
                getApplicationEventMulticaster().addApplicationListener(listener);
        }

        // Do not initialize FactoryBeans here: We need to leave all regular beans
        // uninitialized to let post-processors apply to them!
    //得到所有的监听器
        String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
    //将每个监听器添加到事件派发器中;
        for (String listenerBeanName : listenerBeanNames) {
                getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
        }
```

```
        // Publish early application events now that we finally have a multicaster...
    //得到之前产生的事件
        Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
        this.earlyApplicationEvents = null;
        if (earlyEventsToProcess != null) {
                for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
            //派发之前步骤产生的事件
                        getApplicationEventMulticaster().multicastEvent(earlyEvent);
                }
        }
}
```

11、finishBeanFactoryInitialization(beanFactory);初始化所有剩下的单实例bean； 1、beanFactory.preInstantiateSingletons();初始化后剩下的单实例bean 1）、获取容器中的所有Bean，依次进行初始化和创建对象有些bean已经被创建过了例如:invokeBeanFactoryPostProcessors(beanFactory)过程就已经创建过了 2）、获取Bean的定义信息；RootBeanDefinition 3）、Bean不是抽象的，是单实例的，不是懒加载； 1）、判断是否是FactoryBean；是否是实现FactoryBean接口的Bean； 2）、不是工厂Bean。利用getBean(beanName);创建对象 实际上就是使用bean工厂中创建对象(具体的过程在我的XmlBeanFactory源码解析(下)中做了更详细的说明) 0、getBean(beanName)； ioc.getBean(); 1、doGetBean(name, null, null, false); 2、先获取缓存中保存的单实例Bean。如果能获取到说明这个Bean之前被创建过（所有创建过的单实例Bean都会被缓存起来） 从private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);获取的 3、缓存中获取不到，开始Bean的创建对象流程； 4、标记当前bean已经被创建 -> markBeanAsCreated(beanName) 防止对线程重复创建 5、获取Bean的定义信息； 6、【获取当前Bean依赖的其他Bean;如果有按照getBean()把依赖的Bean先创建出来；】 7、启动单实例Bean的创建流程； 1）、createBean(beanName, mbd, args); 2）、Object bean = resolveBeforeInstantiation(beanName, mbdToUse);让BeanPostProcessor先拦截返回代理对象； 【InstantiationAwareBeanPostProcessor】：提前执行； 先触发：postProcessBeforeInstantiation()； 如果有返回值：触发postProcessAfterInitialization()； 3）、如果前面的InstantiationAwareBeanPostProcessor没有返回代理对象；调用4） 4）、Object beanInstance = doCreateBean(beanName, mbdToUse, args);创建Bean 1）、【创建Bean实例】；createBeanInstance(beanName, mbd, args); 利用工厂方法或者对象的构造器创建出Bean实例； 2）、applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName); 调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition(mbd, beanType, beanName); 3）、【Bean属性赋值】populateBean(beanName, mbd, instanceWrapper); 赋值之前： 1）、拿到InstantiationAwareBeanPostProcessor后置处理器；postProcessAfterInstantiation()； 2）、拿到InstantiationAwareBeanPostProcessor后置处理器；postProcessPropertyValues()； =====赋值之前：===== 3）、应用Bean属性的值；为属性利用setter方法等进行赋值；applyPropertyValues(beanName, mbd, bw, pvs); 4）、【Bean初始化】initializeBean(beanName, exposedObject, mbd); 1）、【执行Aware接口方法】invokeAwareMethods(beanName, bean);执行xxxAware接口的方法BeanNameAware\BeanClassLoaderAware\BeanFactoryAware 2）、【执行后置处理器初始化之前】applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);BeanPostProcessor.postProcessBeforeInitialization（）; 3）、【执行初始化方法】invokeInitMethods(beanName, wrappedBean, mbd); 1）、是否是InitializingBean接口的实现；执行接口规定的初始化； 2）、是否自定义初始化方法； 4）、【执行后置处理器初始化之后】applyBeanPostProcessorsAfterInitialization BeanPostProcessor.postProcessAfterInitialization()； 5）、注册Bean的销毁方法； 5）、将创建的Bean添加到缓存中singletonObjects； ioc容器就是这些Map；很多的Map里面保存了单实例Bean，环境信息。。。。 3)、所有Bean都利用getBean创建完成以后； 检查所有的Bean是否是SmartInitializingSingleton接口的；如果是；就执行afterSingletonsInstantiated();

```
public void preInstantiateSingletons() throws BeansException {
        if (this.logger.isDebugEnabled()) {
                this.logger.debug("Pre-instantiating singletons in " + this);
        }

        // Iterate over a copy to allow for init methods which in turn register new bean
```

```
definitions.
        // While this may not be part of the regular factory bootstrap, it does otherwise work fine.
        //获取容器中的所有Bean，依次进行初始化和创建对象有些bean已经被创建过了例
如:invokeBeanFactoryPostProcessors(beanFactory)过程就已经创建过了
    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

        // Trigger initialization of all non-lazy singleton beans...
        for (String beanName : beanNames) {
        //获取Bean的定义信息；RootBeanDefinition 其过程在xmlBeanFacory下篇中分析过了
                RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        //Bean不是抽象的，是单实例的，不是懒加载
                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                        //判断是否是FactoryBean；是否是实现FactoryBean接口的Bean；
            if (isFactoryBean(beanName)) {
                                final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                                boolean isEagerInit;
                                if (System.getSecurityManager() != null && factory instanceof
SmartFactoryBean) {
                                        isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
                                                @Override
                                                public Boolean run() {
                                                        return ((SmartFactoryBean<?>)
factory).isEagerInit();
                                                }
                                        }, getAccessControlContext());
                                }
                                else {
                                        isEagerInit = (factory instanceof SmartFactoryBean &&
                                                        ((SmartFactoryBean<?>)
factory).isEagerInit());
                                }
                                if (isEagerInit) {
                                        getBean(beanName);
                                }
                        }
                        else {
                /*
                不是工厂Bean。利用getBean(beanName);创建对象
                实际上就是使用bean工厂中创建对象(具体的过程在我的XmlBeanFactory源码解析(下)中做了更详细的说明)
                */
                                getBean(beanName);
                        }
                }
        }

        // Trigger post-initialization callback for all applicable beans...
        for (String beanName : beanNames) {
                Object singletonInstance = getSingleton(beanName);
        /*
        所有Bean都利用getBean创建完成以后；
                检查所有的Bean是否是SmartInitializingSingleton接口的；如果是；就执行
afterSingletonsInstantiated();
        */
                if (singletonInstance instanceof SmartInitializingSingleton) {
                        final SmartInitializingSingleton smartSingleton =
```

```
(SmartInitializingSingleton) singletonInstance;
                        if (System.getSecurityManager() != null) {
                                AccessController.doPrivileged(new PrivilegedAction<Object>() {
                                        @Override
                                        public Object run() {
                                                smartSingleton.afterSingletonsInstantiated();
                                                return null;
                                        }
                                }, getAccessControlContext());
                        }
                        else {
                                smartSingleton.afterSingletonsInstantiated();
                        }
                }
        }
}
```

12、finishRefresh();完成BeanFactory的初始化创建工作；IOC容器就创建完成；

```
        1）、initLifecycleProcessor();初始化和生命周期有关的后置处理器；LifecycleProcessor
                        默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】；如果没有new
DefaultLifecycleProcessor();
                        加入到容器；
    写一个LifecycleProcessor的实现类，可以在BeanFactory
                        void onRefresh();
                        void onClose();
        2）、     getLifecycleProcessor().onRefresh();
                        拿到前面定义的生命周期处理器（BeanFactory）；回调onRefresh();
        3）、publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；
        4）、liveBeansView.registerApplicationContext(this);
```

```
protected void finishRefresh() {
    // Initialize lifecycle processor for this context.
    //initLifecycleProcessor();初始化和生命周期有关的后置处理器
    initLifecycleProcessor();

    // Propagate refresh to lifecycle processor first.
    //getLifecycleProcessor().onRefresh();
    getLifecycleProcessor().onRefresh();

    // Publish the final event.
    //publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；
    publishEvent(new ContextRefreshedEvent(this));

    // Participate in LiveBeansView MBean, if active.
    //liveBeansView.registerApplicationContext(this);
    LiveBeansView.registerApplicationContext(this);
}
```

总结说明:ApplicationContext其实也是以BeanFactory为核心然后里面放入更多的组件，如果环境信息 事件处理器(后续会新开一篇博客专门讲这个问题)等等